SENG201 Virtual Pets Assignment

Ollie Chick – 67683982 and Samuel Pell – 57046549

There are several different design patterns used in this implementation of the Virtual Pets game: prototyping for item creation, observer objects for interclass communication, and model-view-controller (MVC) for the user interface. The observer pattern allows the MVC to be successfully implemented. This is in addition to the interfaces, and inheritance used both for implementing these patterns and for good object-oriented design.

The prototyping design pattern is used to allow flexibility in toy and food creation. The item data is stored in csv files; at runtime, the program loads these files and creates prototype objects that are then stored in GameEnvironment for later use. These items are copied whenever the user buys them, and the copy is placed in the player's inventory. This was chosen so that file input/outputfor the toys and foods was limited to the beginning of the game and means the game always has a copy of all items.

Since the observer design pattern is so intimately tied up with the MVC implementation, they will be discussed together. MVC describes the general architecture of the program. The model is comprised of GameEnvironment, Pet, the individual pet classes, Item, Toy, and Food. The view is comprised of all panels, and the controller is GUIMain. The observer pattern allows the view and controller to communicate. This pattern is implemented by the Observer and Observable interfaces. This was chosen as it was the easiest way to ensure that the GUI is modular and easy to replace during development and in the future. The model and the controller do not communicate via this pattern. Instead an instance of GameEnvironment is stored as a class attribute and is the main interface between the model and the controller.

Inheritance was used in the model in a limited fashion. As can be seen in the UML diagram, it is only Toy, Pet, and the Panels which employ any form of inheritance (except for inheriting from Object, which all Java classes do by default). Alpaca, Cat, Dog, Goat, Horse, and PolarBear act only as wrapper classes for Pet. This was chosen as the pets all performed the same actions (albeit with different values). This also allows them to be stored in a single ArrayList. The wrapper classes were created for programmer usability. Food and Toy both employ true inheritance and are children of Item. This is as they are displayed together in the store, but have slightly different functionality, which requires them to have different method sets. In the view, however, inheritance is prevalent:all of the panels are subclasses of JPanel. This is so that the panels have all the required functionality for their use.

The only external interface that was implemented was Comparable. This was implemented by Player so that a list of players could be sorted by score. This reduced the logic necessary for ranking the players at the end of the game.

Collections are used throughout the Virtual Pets game to store and manage foods, toys, items in general, players, pets, strings, and observers. Two HashMaps, from string to food/toy, were used to store the prototypes of the foods and toys for the store to recall. During setup, the players are added to an ArrayList in the game environment, and their pets are added to ArrayLists in the player objects. The pets are also added to another ArrayList which is used to check if there are any live pets left. Used names were stored as an ArrayList of strings, which was used to determine if there were any duplicates during setup. The store used an ArrayList to store the names of food and toys to display for sale. When the user buys a food, a shallow copy of the food prototype (retrieved from the string to food HashMap) is added to the user's food stock (which is an ArrayList), and when the user buys a toy, a clone of the toy prototype (retrieved from the string to toy HashMap) is added to the user's toy list (which is an ArrayList). One of these collections is displayed to the user when they want to feed or play with their pet. At the start of each day, the foods in each player's food stock is

added to a new ArrayList of food, except any fresh fish, which is replaced with off fish. The player's food stock is then set to this new collection.

Our unit tests cover all of the model, which is the only thing that can be testing with unit tests. The GUI elements cannot be tested via unit tests as there is no objective measure of success. The controller cannot be tested as it is too complicated to be encapsulated by a unit test.

Thoughts and retrospective

This assignment was very interesting: though initially it looked relatively simple, it was a hard problem to solve. The main problem was the scale of the project.

If the program was to be rewritten, improvements could be made. One major flaw is the way pet selection is handled in the main game loop: cycles through each player's pets one after another, rather than allowing random pet selection. The choice to cycle through pets was inherited from the command line version of the program and not rethrought for the GUI implementation decided early on in the GUI version to mimic the command line version. This was a mistake, as choosing this option lead to complicated and unscalable control logic around choosing the next pet, and what to do if they die pet death. Unfortunately, by the time this fact was realised, it was too late to change itthis.

The second major flaw is that all of the GUI uses absolute layout, so it. This means that means that the GUI doesn't scale well. For example, if the user has text larger than the default for Windows or Linux Mint, some of the labels will get cut off. This means the game is not very accessible to the visually impaired or those using tablet computers where larger text is needed. This also means that window scaling is not an option. However, this couldThis however can be changed easily as the GUI panels are very modular and exist separately to both the model and the view.

One major strength of the program is its scalability in creating more toys, foods, and pets. Creating more types of toys and foods is as simple as adding another line to the corresponding csv file. This means that it would also be relatively easy to add functionality for the user to create their own foods and toys. Creating new pets would be slightly more complicated, as radio buttons for the new species and three images would have to be created and implemented. This could be made more scalable by using one list of species names. Radio button creation could iterate through this list and image generation could pull up the image with a filename related to the species name, e.g. `fileName = "img/" + pet.getSpecies + "Toilet.png"`, rather than from a hardcoded switch statement.

Even with those flaws, the program satisfies the brief and everything else went well; however, next time we would both use a GUI layout other than absolute. The first flaw mentioned could not have been easily predicted so there was no way to avoid it. One potential solution would be to test earlier in the process. This would have caught the flaw earlier, and the section could have been rewritten.

Contributions

Ollie Chick - 50%

I was responsible for the Player and Pet classes, as well as the main game loop and the implementation of the command line version. I also designed the csv files used by the program to retrieve data about the pets, foods, and toys.

Samuel Pell - 50%

I was responsible for Item, Food, and Toy early in the project. I also designed the setup process for the game, implemented the view and its interaction with the controller, and handled file IO. All of the images in the game were sourced by me, sized down, and edited if necessary.

Signed:  (Ollie Chick) and  (Samuel Pell)